METAOBJ: Very High-Level Objects in METAPOST

Denis Roegel
LORIA
Campus scientifique
BP 239
54506 Vandœuvre-lès-Nancy cedex
FRANCE
roegel@loria.fr
http://www.loria.fr/~roegel/

Abstract

This paper presents the METAOBJ system and its features for the implementation of very high-level objects within METAPOST.

Introduction

In recent years, METAPOST has become a very interesting and powerful tool for graphics, especially in the context of TEX, for which it is tailored. With METAPOST, the user writes a program describing a drawing. As a full-fledged programming language is at hand, it is possible to automate a number of tasks, and make use of functions for recurring parts in drawings. Therefore, one notable strength of METAPOST is its control of the drawings and the ease with which it becomes possible to ensure homogeneity. METAPOST appears also extremely useful in applications where drawings are generated automatically.

However, most applications to date are using only very simple METAPOST features. The power of METAPOST appears to have not yet been fully explored. One of the ideas which seemed worth exploring was the manipulation of objects and functions on objects within METAPOST. This was the initial aim of METAOBJ, the system which is presented here. METAOBJ is a very large extension of METAPOST, and the most complete (though not complete) description can be found in "The METAOBJ tutorial and reference manual" (Roegel, 2001).

In this article, we will present briefly the usual low-level way of drawing within METAPOST, as well as its shortcomings. We will then describe a functional approach to drawing, then show how objects can be implemented. An example will follow, before a more general overview of METAOBJ.

Low-level drawing in METAPOST

Several kinds of low-level drawings are possible in METAPOST.

Discardable drawings First, there are drawings which can be immediately used and which require no memorization. We call these drawings "discardable". An obvious example is drawing a square with:

```
draw (0cm,0cm)--(1cm,0cm)--
     (1cm,1cm)--(0cm,1cm)--cycle;
```

Many squares can be drawn that way, but if the user wants two squares with the same sizes, he/she must make sure that the drawing function uses the same values.

Hence, discardable drawings are sufficient for simple tasks, but as soon as the application becomes complex, they exhibit many drawbacks.

Memorizable drawings After discardable drawings, we have memorizable drawings. In this case, functions (or macros) are introduced and the use of these functions makes it much easier to obtain homogeneous drawings and to ensure that certain conditions are met. Functions can be called with parameters and producing two identical drawings only affords giving identical values to certain parameters. For instance, a more elaborate version of the square can be obtained with the following function:

```
def draw_Square(expr p,l,a)=
  draw p--(p+l*dir(a))--
       (p+l*dir(a)+l*dir(a+90))--
       (p+l*dir(a+90))--cycle;
enddef;
```

We now have a way to reuse drawing instructions. The `draw_Square` macro could even be enriched and perform other tasks than merely drawing a square.

The `draw_Square` macro actually contains the definition of the square. However, this macro does too much in that it also draws the square. We could

define a macro only defining the square as a path, for instance as:

```
def Square(expr p,l,a)=
  (p--(p+l*dir(a))
    --(p+l*dir(a)+l*dir(a+90))
    --(p+l*dir(a+90))--cycle)
enddef;
```

In that case, since `Square` returns a closed path, we can do several things with it. We can draw it with

```
draw Square(origin,1cm,50);
```

or we can fill it:

```
fill Square(origin,1cm,50);
```

Other options are also available.

However, even though we have a nice encapsulation of a square, such a definition may not be convenient for other "objects." For instance, if we want to define a function for a square with a double frame, we would not be able to define it in the same way, because a double frame is not a METAPOST path. In order to draw such an object, a special function may need to be introduced, for instance:

```
def drawDoubleSquare(expr p,l,a)=
  draw Square(p,l,a);
  draw Square(p-.5mm*dir(a+45),l+1mm,a);
enddef;
```

The point is that different objects are treated differently, which makes it cumbersome. A simple square can be drawn with `draw`, but a double square uses a special function.

Moreover, there are other underlying problems in the two approaches we have described, namely that the object itself is difficult to grasp. Even though the content of a double square is the definition of the `drawSquare` macro, the automatic analysis or extraction of parts of the definition is not easy. In the case of a simple square, the path could be memorized easily and all its points could be obtained by standard METAPOST functions, but other objects would need other treatments. This would result in different and non-homogeneous handling.

One might argue that it is not necessary to go beyond drawing, but in fact, there are examples where parts of objects need to be accessed once they are drawn; for instance, to specify certain non-global alignments.

We will now describe how objects can be accessed in an homogeneous way.

A functional approach to drawing

Before describing the implementation of objects, we will give an abstract description of the main features of an object. An object o is a structure with a name (o), points (which have names), linear equations between points, paths, and several other features that will be described later.

Objects will be of different types (classes), and to each type will correspond certain functions. Not all functions make sense with all types of objects. However, there is a category of "standard" objects to which the main functions apply. For instance, some of the functions are linear transformations: scaling, rotating, slanting or reflecting. When an object o is given, it can be rotated 90 degrees counterclockwise with `rotateObj(o,90)`.

Some of the functions are general (static) and not attached to an object, but others may be attached to an object and can be viewed as methods of the object. For instance, every time an object is drawn (with `drawObj(o)`), it is in fact a method of o, or more precisely a method defined in o's class which is called.

The functional approach to drawing is interesting because functions can return objects. By default, `rotateObj(o,90)` only rotates o, but cannot be composed. This corresponds to the imperative object-oriented construction of an object. It is object-oriented because o is an object and because a method of o is called. It is imperative because the operation is self-contained. In order to use a functional approach, METAOBJ provides a functional variant of `rotateObj` named `rotate_Obj`. It is then possible to write

```
rotate_Obj(rotate_Obj(o,60),30)
```

and this will return the initial object rotated $30+60$ degrees.

New functions can be written and added to the standard library. These functions will then take their power from the reflexion capability of METAOBJ, namely the possibility for the function to explore the object to which it applies, making therefore virtually anything possible.

The structure of an object

The main novelty of METAOBJ is the fact that it is a system which keeps track of all of an object's features. It is possible to reflect on an object's structure, and it is actually also possible to have functions creating objects, and even classes of objects, given certain parameters. The possibilities are endless and have hardly been explored so far.

An object has a name and all its direct components form a tree of variables, all starting with the object name. There is unfortunately no easy way to traverse such a tree of variables in METAPOST, and there are therefore special variables which keep

Attribute	Type	Default	Description
pointlist_	string	""	list of points
pairlist_	string	""	list of pairs (non-movable points)
pointarraylist_	string	""	list of arrays of points
subarraylist_	string	""	list of arrays of subobjects
stringarraylist_	string	""	list of arrays of strings
colorarraylist_	string	""	list of arrays of colors
picturearraylist_	string	""	list of arrays of pictures
transformarraylist_	string	""	list of arrays of transforms
booleanarraylist_	string	""	list of arrays of booleans
numericarraylist_	string	""	list of arrays of numerics
pairarraylist_	string	""	list of arrays of pairs (non-movable points)
points_in_arrayslist_	string	""	enumeration of all (movable) points of all arrays (of movable points)
picturelist_	string	""	list of pictures
numericlist_	string	""	list of numerics
sublist_	string	""	list of subobjects
subobjties_	string array		subobj tying equations (1 string/subobject)
nsubobjties_	numeric	0	number of subobjties
code_	string	""	code of an object
extra_code_	string	""	extra code of an object
ctransform_	transform	identity	current transform of that object

Table 1: Standard attributes of an object

track of what is in an object. We call these variables the *attributes* of an object. The complete list of attributes is given in table 1.

```
ptre=100
ptre.numericlist_="dx,dy,nst"
ptre.nst=2
ptre.ctransform_=(0,0,1,0,0,1)
```

Figure 1: Object structure: general data

As shown in figure 1, an object has a unique identifying number. Here, it is 100. Different numbers correspond to different objects. In particular, subobjects will also have their own identifying number.

Each object can store numerical values in the string `numericlist_`. There are three here, `dx`, `dy`, and `nst` (figure 1), but only `nst` is defined. This is the number of subtrees (2 in this example).

`ctransform_` records the current transform of the object, and it is initially the identity.

An object also has points (figure 2), the list of which is given in the string `pointlist_` which is a standard attribute of the object. It has a list of subobject references (figure 3), the list of which is given as a string `sublist_`. Each subobject is actually only given as a string. The subobject is not literally part of the object (an object can be part of several other objects).

The `ptre` object belongs to the `PTree` class (this information is not part of the object, but can be obtained by an external table) and it contains four subobjects. Each subobject has a corresponding string: `conc` (conclusion), `subt` (subtrees), `lr` (left rule), `rr` (right rule). The value of the string (figure 3) was generated automatically by `newobjstring_`. The fancy names avoid name clashes with user-defined objects.

The structure of `ptre` was obtained with

```
showObj ptre;
```

but this does not show the structure of all subobjects. We could define a function showing recursively the whole structure of an object, but currently the user must call `showObj` on each subobject `_______zu`, etc.

It is possible to go to great depth in an object. Even when there are `picture`s, we can find what is inside using the `for ... within` construct.

Each subobject has an associated tying function. This function attaches a subobject to the main object and is used when linear transformations are

```
ptre.pointlist_="ne,nw,sw,se,n,s,e,w,c,
                 ine,inw,isw,ise,in,is,ie,iw,ic,ledge,redge,lstart,lend"
ptre.c=(0,287.17845)
ptre.n=(0,312.73784)
ptre.s=(0,261.61906)
ptre.e=(95.02467,287.17845)
ptre.w=(-95.02467,287.17845)
ptre.ne=(95.02467,312.73784)
ptre.se=(95.02467,261.61906)
ptre.nw=(-95.02467,312.73784)
ptre.sw=(-95.02467,261.61906)

ptre.ic=(0,283.46451)
ptre.in=(0,308.30998)
ptre.is=(0,258.61905)
ptre.ie=(98.02467,283.46451)
ptre.iw=(-98.02469,283.46451)
ptre.ine=(98.02467,308.30998)
ptre.ise=(98.02467,258.61905)
ptre.inw=(-98.02469,308.30998)
ptre.isw=(-98.02469,258.61905)

ptre.ledge=(-72.57094,258.61905)
ptre.redge=(92.90178,258.61905)
ptre.lstart=(-51.91089,292.35118)
ptre.lend=(65.57219,292.35118)
```

Figure 2: Object structure (cont'd): points

```
ptre.sublist_="subt,lr,rr,conc"

ptre.conc="_______zu"
ptre.subt="_______zh"
ptre.lr="_______zs"
ptre.rr="_______zt"

ptre.nsubobjties_=4

ptre.subobjties_1="vardef tie_function_@#(expr $)=q_1=obj(@#subt).ne;
                   transformObj(obj(@#subt))($);
                   @#ne-obj(@#subt).ne=(p_1-q_1) transformed $;enddef;"
ptre.subobjties_2="vardef tie_function_@#(expr $)=q_2=obj(@#lr).ne;
                   transformObj(obj(@#lr))($);
                   @#ne-obj(@#lr).ne=(p_1-q_2) transformed $;enddef;"
ptre.subobjties_3="vardef tie_function_@#(expr $)=q_3=obj(@#rr).ne;
                   transformObj(obj(@#rr))($);
                   @#ne-obj(@#rr).ne=(p_1-q_3) transformed $;enddef;"
ptre.subobjties_4="vardef tie_function_@#(expr $)=q_4=obj(@#conc).ne;
                   transformObj(obj(@#conc))($);
                   @#ne-obj(@#conc).ne=(p_1-q_4) transformed $;enddef;"
```

Figure 3: Object structure (cont'd): subobjects

```
ptre.code_="@#se-@#sw=@#ne-@#nw;xpart(@#se-@#ne)=0;ypart(@#se-@#sw)=0;
            @#n=.5[@#ne,@#nw];@#s=.5[@#se,@#sw];@#e=.5[@#ne,@#se];
            @#w=.5[@#nw,@#sw];@#c=.5[@#n,@#s];@#ine=@#ne;@#inw=@#nw;
            @#isw=@#sw;@#ise=@#se;@#in=@#n;@#is=@#s;@#ie=@#e;@#iw=@#w;
            @#ic=@#c;
            xpart(.5[obj(@#conc).ledge,obj(@#conc).redge])=
               .5[xpart(obj(@#subt).s)-62.07625,xpart(obj(@#subt).s)+55.40683];
            ypart(obj(@#subt).s-obj(@#conc).n)=5.66928;
            ypart(@#n-obj(@#subt).n)=0;ypart(obj(@#conc).s-@#s)=0;
            @#ledge=obj(@#conc).sw;@#redge=obj(@#conc).se;
            ypart(@#lstart)=ypart(@#lend)=ypart(obj(@#conc).n)+2.83464;
            xpart(@#lstart)=xpart(obj(@#subt).c)-62.07625+0;
            xpart(@#lend)=xpart(obj(@#subt).c)+55.40683+0;
            xpart(@#e)-xpart(obj(@#subt).e)=+0;
            xpart(obj(@#subt).w)-xpart(@#w)=+20.33072;
            @#lstart-(rdistl,0)=obj(@#lr).e;"
```

Figure 4: Object structure (cont'd): equations

applied to an object. Having four subobjects, we have therefore four such functions, `subobjties_1`, ..., `subobjties_4`. These functions are stored as strings.

And finally, all the equations defining the initial state of the object are stored in the `code_` string (figure 4). These equations are used whenever an object is reset.

An object can contain more information, but every time there is some variable, the name of this variable must also appear in some list, because this is the only way to achieve duplication (cloning). We can only know what is inside an object if we constantly keep track of it. This also explains why special functions should be used to define variables. "`pair`" would not be enough to record the name of a pair. Instead, "`ObjPoint`" should be used.

An object definition example

The simplest of all classes is the `EmptyBox`. An `EmptyBox` is an empty rectangle, normally with no frame. Its only purpose is to take some space. For instance, it is useful in order to change the spacing between leaves of a tree, when the spacings are not all identical. The constructor looks like:

```
vardef newEmptyBox@#(expr dx,dy)
     text options=
  ExecuteOptions(options);
  assignObj(@#,"EmptyBox");
  StandardInterface;
  ObjCode StandardEquations,
   "@#ise-@#isw=(" & decimal dx & ",0)",
   "@#ine-@#ise=(0," & decimal dy & ")";
enddef;
```

It is called with two dimensions, which are the sides of the rectangle. It should be noticed that the values of `dx` and `dy` can be negative and this can produce some special effects.

The name of the object created is the suffix represented here as `@#`. The creation of box `b` would be done with:

```
newEmptyBox.b(2cm,1cm);
```

The syntax is therefore similar to the one found in the `boxes.mp` package.

The constructor also exhibits some features related to the options mechanism. Every constructor can have options modifying its behavior. The options are given as the last parameters of the constructor and are used in a call to `ExecuteOptions`. Each object decides which option it honors and how. The `EmptyBox` doesn't have many options, but it is still possible to draw its frame with a different thickness or to fill the box. Therefore, the `drawEmptyBox` function is (with slight simplifications):

```
def drawEmptyBox(suffix n)=
  if show_empty_boxes:
    drawFramedOrFilledObject_(n);
  fi;
  drawMemorizedPaths_(n);
enddef;
```

This macro is simple: depending on the global variable `show_empty_boxes` (often used for debugging), the empty boxes are shown or not. If they are shown, they are either filled or merely drawn. The `drawFramedOrFilledObject_` takes care of the various cases. If they are filled, they are filled with a color that can be given as an option.

If the object is filled, the "bounding path" of the object is used, and it is given by the function `BpathEmptyBox`. Like the function `drawObj` which calls `drawEmptyBox`, the `BpathObj` function actually

calls `BpathEmptyBox`. Each object must declare its "bounding path" function. For `EmptyBox`, we have

```
def BpathEmptyBox(suffix n)=
  StandardBpath(n)
enddef;
```

The standard bounding path provided by the `StandardBpath` function is merely the path `n.inw -- n.isw -- n.ise -- n.ine --cycle`. This path uses the "inner interface" so that the drawing of the object does not depend on artificial changes to its bounding box.

Overview of METAOBJ classes

METAOBJ defines standard object classes, which can then be instantiated. The standard library includes:

- basic objects: these objects are not containers and appear therefore at the leaves of a structure hierarchy:
  - `EmptyBox`: a rectangle with a given size;
  - `HRazor` and `VRazor`: this is a degenerated `EmptyBox`;
  - `RandomBox`;
- basic containers: such objects can contain any other object, and have varying shapes for the frame:
  - `Box`: a (usually) rectangular frame;
  - `Polygon`: a polygonal frame, whose number of sides can be parameterized;
  - `Ellipse`: an elliptic frame, whose shape can be parameterized;
  - `Circle`: a special case of `Ellipse`;
  - `DBox`: a box, with a doubled frame;
  - `DEllipse`: an ellipse, but with a doubled frame;
- box alignment constructors:
  - `HBox`
  - `VBox`
- recursive objects and fractals:
  - `RecursiveBox`: a box, containing a box, containing a box, ...
  - `VonKochFlake`: a well-known fractal;
- trees:
  - `Tree`: general trees;
  - `PTree`: proof trees;
- matrices: `Matrix`

A complex example

Figure 5 shows a much more elaborate example created with METAOBJ. The code provided should be easy to follow. First, a box named a is created, with the text "a". This is the square box "a" within the big circle, but at that point it is located nowhere. The box is actually "floating". Similarly, two ellipses are created, named b and c. The latter appears differently because the constructor (`newEllipse`) was called with several options which change the ellipse. From a, b, and c we create a tree, specifying the color of the arcs (which are arrows by default). The name of the tree is t and it is then put inside a box named aa. The new box is given round corners. This boxed tree is floating, until `aa.c=origin;` is executed. This is similar to how boxes are positioned with John Hobby's `boxes` package.

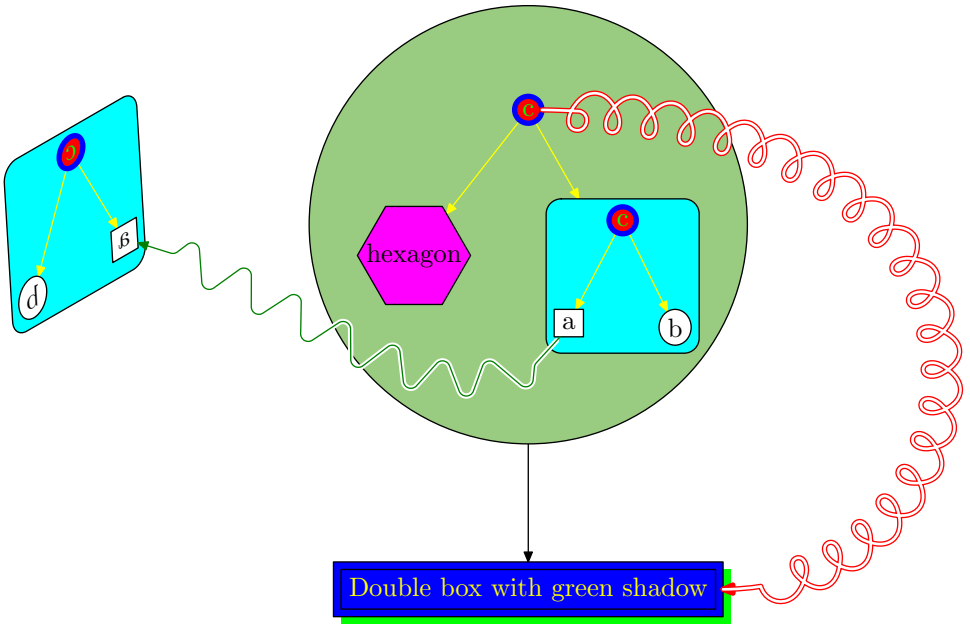
Later, another tree is built, and the whole tree is put inside a circle. This circle then becomes the root of a new tree and this tree is drawn with the call to `drawObj(nt)`. This automatically draws all components recursively. A spring then connects an ellipse (xc) to a double box (db). The spring (`nccoil`) has various options which were taken from PSTricks. It connects two named components without referring to the whole tree. However, it would have been possible to extract these two objects from the whole tree.

The rounded box (aa) is then duplicated and a new floating object dt is created. This copy is reflected, slanted and rotated. All these operations are applied recursively to all components of dt. The new object is drawn after having been put in place with `dt.c=nt.c-(6cm,-1cm)`. Finally, a zigzag connection is added between the former a object and its copy. The copy of a has an internal name, but we do not know it, and therefore we access the copy with the `treepos` function specifying the first child of dt.

Conclusion and related work

METAOBJ makes it possible to define and manipulate objects. The standard functions consider that the objects are rigid and can be combined in various ways. No matter how an object has been built, its structure is still easily accessible and open to introspection.

It is of course easy to add new objects and we have only provided a few. It is also easy for the programmer to write special packages manipulating special graphical formalisms, such as UML, etc.




```
input metaobj

beginfig(1);
  newBox.a("a");
  newEllipse.b("b");
  newEllipse.c("c") "filled(true)", "fillcolor(red)","picturecolor(green)",
                    "framecolor(blue)","framewidth(2pt)";
  newTree.t(c)(a,b) "linecolor((1,1,0))";
  newBox.aa(t) "filled(true)", "fillcolor((0,1,1))","rbox_radius(2mm)";
  aa.c=origin;
  newHexagon.xa("hexagon") "fit(false)","filled(true)","fillcolor((1,0,1))";
  newEllipse.xc("c") "filled(true)", "fillcolor(red)","picturecolor(green)",
                   "framecolor(blue)","framewidth(2pt)";
  newTree.xt(xc)(xa,aa) "linecolor((1,1,0))";
  newCircle.xaa(xt) "filled(true)", "fillcolor((.6,.8,.5))";
  newDBox.db(btex Double box with green shadow etex)
        "shadow(true)", "shadowcolor(green)",
        "filled(true)","fillcolor(blue)","picturecolor((1,1,0))";
  newTree.nt(xaa)(db);
  drawObj(nt);
  nccoil(xc)(db) "angleA(0)","angleB(180)",
    "coilwidth(5mm)","linetension(0.8)","linecolor(red)",
    "doubleline(true)","posB(e)";
  duplicateObj(dt,aa);
  reflectObj(dt,origin,up);
  slantObj(dt,.5);
  rotateObj(dt,30);
  dt.c=nt.c-(6cm,-1cm);
  drawObj(dt);
  nczigzag(a)(treepos(obj(dt.sub))(1))
    "angleA(-120)","coilwidth(7mm)","linecolor(.5green)","linearc(1mm)",
    "border(2pt)";
endfig;
```

Figure 5: A complex example and its source code.

It should also be possible to extend the concept of interface to other kinds of interfaces and to introduce objects that are not completely rigid. It is possible to use the complete object structure to implement tree layout algorithms. We could also have objects whose shape depends on parameters and on their location in a drawing. More elaborate operations could also be implemented if the history of an object was known. This is currently not the case, and causes limitations in certain experimental features such as the reset feature.

Other refinements include support for layers. This feature is currently not included in METAOBJ, but could be added in the future.

METAOBJ can be used as a replacement to the `boxes.mp`, `rboxes.mp`, and `fancybox.sty` packages and to many features found in PSTricks. Several features of METAOBJ, for instance connections, can be used without a reference to objects.

METAOBJ was especially influenced by my earlier work on animations (Roegel, 1997) where an object notion was introduced. The 3D objects were very primitive, but they provided many useful ideas. Several objects have been influenced from counterparts in various packages. This is especially true for rectangular and elliptic boxes. PSTricks provided many ideas and the connection functions are entirely borrowed from that package. Only PSTricks's user documentation was used, not the implementation details (van Zandt and Girou, 1994), though similarities can be observed. There has also been some work by Denis Girou on high-level objects in PSTricks (Girou, 1995), but this work is not really relevant to what we have done. Another work we didn't use was Kristoffer Rose's work on high level 2-dimensional graphics (Rose, 1997). The proof tree class was influenced by a LaTeX package I wrote in 1993 and which was never released.

Several systems explore similar ideas, though they did not influence METAOBJ. One is "Functional METAPOST," a system providing a Haskell layer to specify drawings abstractly (Korittky, 1998; Kuhlmann, 2001). The drawback of that approach is that one needs a Haskell compiler and of course one needs to learn some of this language. Nevertheless, this work shares many ideas with METAOBJ in that objects are built by applying functions to already existing structures. Another functional approach to picture drawing is FPIC which uses the ML language (Kamin and Hyatt, 1997). A very popular system with many similar ideas is the 2D Java API (Knudsen, 1999). It provides graphical objects which can be manipulated by various transformations and modified in various ways.

The syntax of METAOBJ is admittedly verbose, but it is hoped that a TeX interface will be provided and that it will alleviate a lot of the user's burden.

References

Girou, Denis. "Building high level objects in PSTricks". 1995. Slides presented at TUG'95, St. Petersburg (Florida).

Goossens, Michel, S. Rahtz, and F. Mittelbach. *The LaTeX Graphics Companion: Illustrating documents with TeX and PostScript*. Reading, MA: Addison-Wesley, 1997.

Hagen, Hans. *ConTeXt: the manual*, 2001a.

Hagen, Hans. *MetaFun*, 2001b.

Hobby, John D. "A User's Manual for MetaPost". Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1992. Computing Science Technical Report 162.

Hoenig, Alan. *TeX Unbound: LaTeX & TeX Strategies for Fonts, Graphics, & More*. New York: Oxford University Press, 1998.

Kamin, Samuel N. and D. Hyatt. "A Special-Purpose Language for Picture-Drawing". In *USENIX Conf. on Domain-specific Languages, Santa Barbara*, pages 297–310. 1997.

Knudsen, Jonathan. *Java 2D Graphics*. O'Reilly & Associates, 1999.

Knuth, Donald E. *The METAFONTbook*. Reading, MA: Addison-Wesley, 1986.

Knuth, Donald E. *Digital Typography*, volume 78 of *CSLI Lecture Notes*. CSLI, 1999.

Korittky, Joachim. "Functional METAPOST. Eine Beschreibungssprache für Grafiken". 1998. Diplomarbeit an der Rheinischen Friedrich-WilhelmsUniversität Bonn.

Kuhlmann, Marco. "Functional METAPOST for LaTeX". 2001.

Ohl, Thorsten. "EMP: Encapsulated METAPOST for LaTeX". 1997. Technische Hochschule Darmstadt.

Roegel, Denis. "Creating 3D animations with METAPOST". *TUGboat* **18**(4), 274–283, 1997.

Roegel, Denis. *The METAOBJ tutorial and reference manual*, 2001.

Rose, Kristoffer Høgsbro. ""Very High Level 2-dimensional Graphics" with TeX and XY-pic". *TUGboat* **18**(3), 151–158, 1997.

van Zandt, Timothy. *PSTree user's guide*, 1993a.

van Zandt, Timothy. *PSTricks: PostScript macros for Generic TeX; User's Guide*, 1993b.

van Zandt, Timothy and D. Girou. "Inside PSTricks". *TUGboat* **15**(3), 239–248, 1994.